In Search of Speculative Thread-Level Parallelism

Jeffrey T. Oplinger David L. Heine Monica S. Lam

Computer Systems Laboratory
Stanford University, CA 94305
{jeffop, dlheine, lam}@cs.stanford.edu

Abstract

This paper focuses on the problem of how to find and effectively exploit speculative thread-level parallelism. Our studies show that speculating only on loops does not yield sufficient parallelism. We propose the use of speculative procedure execution as a means to increase the available parallelism. An additional technique, data value prediction, has the potential to greatly improve the performance of speculative execution. In particular, return value prediction improves the success of procedural speculation, and stride value prediction improves the success of loop speculation.

1. Introduction

Researchers have been exploring the use of speculative threads to harness more of the parallelism in general-purpose programs [1][6][8][12][15][17][19]. In these proposed architectures, threads are extracted from a sequential program and are run in parallel. If a speculative thread executes incorrectly, a recovery mechanism is used to restore the machine state. While a superscalar processor can only extract parallelism from a group of adjacent instructions fitting in a single hardware instruction window, a thread-based machine can intelligently find parallelism among many larger, non-sequential regions of a program's execution. Speculative threads can also exploit more parallelism than is possible with conventional multiprocessors that lack a recovery mechanism. Speculative threads are thus not limited by the programmer's or the compiler's ability to find guaranteed parallel threads. Furthermore, speculative threads have the potential to outperform even perfect static parallelization by exploiting dynamic parallelism, unlike a multiprocessor which requires conservative synchronization to preserve correct program semantics.

Several hardware designs have been proposed for this speculative thread-level parallelism (STP) model [1][6][8][12][15][17][19], but so far the speedup achieved on large general-purpose integer code has been limited. However, it is important to note that these experiments evaluated not only the proposed hardware, but also the choices made by the researcher or the compiler as to where to apply speculative execution. The decision on where to speculate can make a large difference in the resulting performance. If the performance is poor, we gain little insight on why it does not work, or whether it is the parallelization scheme or machine model (or both) that should be improved. As a consequence, poor results may not reflect any inherent limitations of the STP model, but rather the way it was applied.

The goal of this paper is to identify potential sources of speculative parallelism in programs. To search through a large space of parallelization schemes effectively, we work with simple machine models and a relatively simple trace-driven simulation tool.

We define an *optimal* STP machine that incurs no overhead in executing the parallel threads and can delay the computation of a thread perfectly to avoid the need to rollback any of the computation. To keep the experiments simple and relatively fast, the STP machine is assumed to be a simple machine in all other aspects. Many different optimizations have previously been proposed to minimize rollbacks, such as adding synchronization operations to the code statically[2] or inserting them dynamically as rollbacks are detected, for example[13]. We can use the optimal machine to derive an upper bound on the performance achievable using any possible synchronization optimizations. The optimal machine serves as an effective tool for filtering out inadequate parallelization techniques, since techniques that do not work well on this machine will not work well on any real machine of a similar design. We vary the resources available on our optimal STP machine in the experiment, supporting 4, 8, or an infinite number of concurrent threads.

We also define a *base* STP machine that is similar to the optimal version but makes no attempt to eliminate rollbacks through synchronization. The machine simply executes the instructions of each thread in sequence; if the data used is found to be stale, and the value was not correctly predicted, the machine restarts the thread. The performance of a particular synchronization scheme should thus fall between the bounds established by the optimal and base machines.

To explore the potential of various parallelization schemes in an efficient manner, we have created a trace-driven simulation tool that can simultaneously evaluate multiple parallelization choices. We also use this tool to collect useful statistical data, providing important insights and explanations of the parallel behavior of the program.

Our experiments have led to the following contributions:

- We found that it is inadequate to exploit only loop-level parallelism, the form of parallelism that is used almost exclusively in many prior studies. Our tool simultaneously evaluates all possible choices of which level in a loop nest to speculate on. Even with optimal loop-level choices for

This research is supported in part by DARPA contracts DABT63-95-C-0089 and MDA904-98-C-A933.

speculation and optimal data dependence synchronization, the speedup obtained is low. This is due to a combination of the limited parallelism of loops in non-numeric code and the time spent in sequentially executed code found outside of the parallelized loops.

- We found that procedures provide a useful source of speculative parallelism. In procedural speculation, another thread speculatively executes the code following the return of the procedure at the same time the procedure body is itself being executed. Procedure boundaries often separate fairly independent computations, and there are many of them in most programs. Thus, procedure calls provide important opportunities for parallelism and can complement loops as a source of thread-level parallelism.
- We also evaluated the potential of using prediction to improve speculative parallelism. In particular, predicting the value returned by a procedure (return value prediction) can greatly reduce the data dependences between the procedure execution and the following computation. Value prediction can also eliminate important data dependence constraints across iterations of speculatively executed loops.

This work suggests that there is significant potential in an STP machine that supports speculation on both loops and procedures. Defining a complete machine design based on this computation model is beyond the scope of this paper. This work serves an important function in showing the limits of certain parallelization schemes, such as parallelizing only loops, and pointing out ideas worthy of further attention such as procedural speculation and value prediction.

The rest of this paper is organized as follows. In Section 2 we describe the STP machine model in more detail, followed by the simulation methodology in Section 3. In Section 4 we present results on using the optimal STP model to exploit loop-level parallelism. We investigate the use of procedural speculation in Section 5. Section 6 contains the results of combining loop-level and procedure-level speculation. In Section 7 we present the performance results of both the optimal and base machines with finite resources. Related work is discussed in Section 8 and we conclude in Section 9.

2. The STP Machine Model

In the STP machine model, the program explicitly specifies when the threads are to be created. To preserve correct sequential execution semantics, the side effects of each speculative thread are saved in a separate speculative state. Each thread can observe all the writes of threads that occur earlier in the sequential execution sequence, use the latest values at each point, and detect dependence violations. There are two forms of dependence violation:

- (true) data dependence. A thread detects a dependence violation if it discovers that an earlier thread has generated a value that it needs after it has already speculatively computed with an outdated value. This is the only form of data dependence violation. Anti-dependences (or storage dependences) do not cause a violation. That is, write operations by a later thread do not affect the values read by an earlier thread, and therefore these operations need not be ordered. Similarly, actions of earlier threads have no effect on a later thread that first writes then reads the same location.
- control dependence. A thread detects a control dependence violation if the control flow of an earlier thread causes the subsequent thread not to be executed at all. For example, an early exit taken by an iteration of a loop would render the speculative execution of all subsequent iterations useless.

If a violation occurs, the processor throws away the speculative state and, in the case of a data dependence violation, restarts the thread; if none occurs, it commits the speculative state when all threads coming earlier in the sequential execution have committed. There have been several proposals that implement this model[3][17][19], which are discussed in Section 8.

Value prediction[11], a concept receiving much recent attention, is particularly relevant to STP as it enables a program to run faster than the dataflow limit determined by data dependences. We examine two simple schemes of value prediction. The last-value prediction (LVP) scheme predicts that the loaded value will be the same as the value obtained from the last execution of that load instruction. The stride-value prediction (SVP) scheme predicts that the loaded value will be the same as the last loaded value plus the difference between the last two loaded values. By using value prediction, a speculative thread need not be rolled back upon detecting a true data dependence violation if the predicted value matches the newly produced value. To find the upper bound of the impact of value prediction on STP, we assume that the machine automatically uses value prediction whenever it attempts to read a value that has not yet been produced. As this is a limit study, we also assume that the machine has a buffer large enough to hold all predicted values needed by any thread.

3. Simulation

We use a trace-driven simulation tool to evaluate the performance of different speculative execution schemes under different machine models. For the sake of simplicity, we assume that each processor takes one cycle to execute each instruction. Also, to focus on the potential of the different parallelization schemes, we assume that the system has a perfect memory hierarchy. All memory accesses can be performed in a single clock and stored data is immediately available to all processing elements in the next cycle. There is no overhead in the creation of threads and no additional cycles are needed to restart a thread once a violation is detected.

The execution of a single instruction per cycle in our simulations raises the issue of how thread-level parallelism interacts with instruction-level parallelism (ILP). Note that the STP model is designed to execute relatively coarse-grain threads, so we expect that the threads in the STP model can still benefit significantly from ILP. The average thread size across our benchmarks was 56 instructions, roughly corresponding to a 224-instruction window on a 4-way machine and a 448-instruction window on an 8-way machine. Still, further studies are necessary to determine the effect of combining STP and ILP.

To explore the STP design space, we vary our simulations based

upon the types of regions speculation is applied to (single-level loops, multi-level loops, procedures, loops and procedures), whether or not dependence synchronization is employed, the maximum number of concurrent threads allowed to execute, and the value prediction policy for memory loads and register reads. The parameters to our simulator are summarized in Table 1 and Table 2.

Table 1: STP Parallelization Schemes

Region(s)	Description
loops	a single loop in each nest is chosen for speculation
multi-level loops	all loops in each nest are chosen for speculation
procedures	procedure bodies execute in parallel with the code following them
loops and procedures	speculation on a single loop in each nest as well as on procedures

Table 2: STP Machine Model Parameters

Parameter	Description
Synchronization Policy	
optimal	all operations delayed optimally to avoid rollback
base	no operations are ever delayed, threads are rolled back as soon as a violation is detected
Resources	
Infinite	no limit on the number of concurrent threads
8-way	resources to execute 8 concurrent threads
4-way	resources to execute 4 concurrent threads
Return Value Prediction	
none	no return value prediction
LRVP	last value prediction of return values whenever procedural speculation is used
LVP	last value prediction of return values whenever procedural speculation is used
SVP	stride value prediction of return values whenever procedural speculation is used
Value Prediction for Memory Loads and Register Reads	
none	no value prediction
LRVP	no value prediction (except for return values as above)
LVP	last value prediction
SVP	stride value prediction

We use the ATOM tool[18] to augment the optimized program binaries and generate a user-level trace that includes events of interest: loads and stores, procedure entries and exits, and loop entries, exits, and iteration advances. (System calls are not captured in the trace.) The simulation clock normally advances one cycle per instruction. Procedure and loop entries signal the potential forking of a speculative thread, depending on the parallelization scheme used. When a speculative thread fork is encountered, the simulation time is stored and the first thread executes. When the later thread(s) from the fork begin to execute, the simulation time is set back to the time of the fork. When a store or register write occurs, the current simulation time is recorded for that memory/register location. Execution continues as normal, except threads are delayed or squashed if they try to read a value that was written at a time greater than the current simulation time. Delays are avoided in the models employing value prediction whenever the values are correctly predicted. Value prediction is implemented by keeping an array of the last two loaded values for each load and register use in the program, and by keeping another array of the last two returned values for each procedure in the program. The value prediction employed is rather idealistic, in that we presume perfect "confidence estimation"—when the prediction is incorrect, the machine simply behaves as if prediction were not employed at all, so misprediction never causes any performance penalty. Additionally, because the simulation is based on the sequential program trace, there is no notion of the value predictor being updated "out of order" as would undoubtedly happen in a real speculative machine. A prediction is always based on the last one or two instances of that instruction in the sequential trace. Thus the performance results with prediction should be considered as an upper bound on the benefit that prediction could provide.

In simulations of finite-processor models, a resource table is used to track usage of the individual processors. A thread must obtain execution cycles from a resource table before it is allowed to execute. Threads closer to the sequential execution are given priority and may preempt lower priority threads in progress if resources are exhausted. Preempted threads are delayed until the next available cycle and are re-executed from the start.

When the actual number of loop iterations is not known *a priori*, a real STP machine would end up wasting some resources executing beyond the end of the loop. To account for the wasted cycles on mispredicted iterations, which do not show up in our sequential trace, our simulator treats the end of a loop as a barrier to parallelization in simulations of finite machines. That is, the machine is not allowed to speculatively execute computation that logically follows the loop in the original sequential execution. No such barrier exists in the procedural case because only one new thread is created at each procedure call.

One final consideration is that the code generated by the compiler for a uniprocessor machine includes many "artificial" dependences. For example, the multiple threads in an STP machine operate on multiple stack frames at the same time, so they need not obey the dependences due to global and stack pointer updates in the sequential program. Similarly, since the threads have separate register files, they are not constrained by the dependences introduced by callee-saved register operations. Our simulator ignores dependences originating from a register restore operation (e.g. the dashed arrow in Figure 1), and instead observes the true dataflow dependence that the save/restore code is preserving (e.g. the solid arrow in Figure 1).

3.1. Benchmarks

We evaluate the performance of our various speculative thread models using the 8 benchmarks from the SPECint95 benchmark suite. Table 3 lists the programs, input sets, and execution characteristics. Throughout the paper, speedups are calculated relative to a single cycle per instruction sequential execution of the program.

All the programs were compiled using the Compaq Alpha `cc` compiler with optimizations using the `-O2` flag. To perform the simulation, we use ATOM to annotate the binaries with

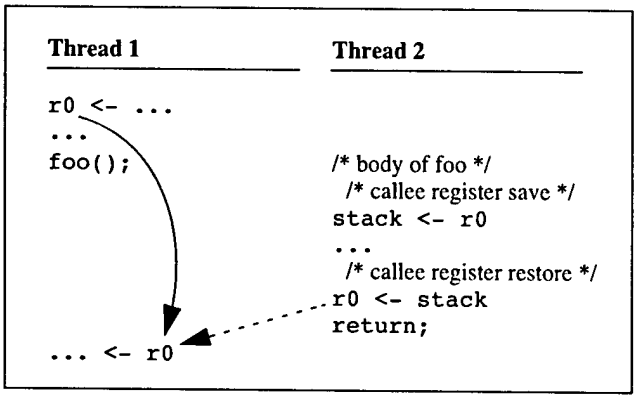

Figure 1. Dependences Induced by Callee-Saved Register Operations

Table 3: Benchmarks Executed

Program	Lines of Code	Input Set	Dynamic Instr	Description
compress	1K	train	47M	File compression
gcc	192K	train	286M	The GNU compiler
go	29K	train	54M	Game of go
ijpeg	28K	train	183M	Image compression
li	7K	train	136M	Lisp interpreter
m88ksim	18K	test	134M	Processor simulator
perl	23K	train (primes.pl)	5M	Perl language interpreter
vortex	52K	train	963M	Database

information such as the entry and exit points of loops as well as locations for `setjmp` and `longjmp` calls. The annotation tool analyzes the binary code directly, extracts control flow graphs from the code and calculates the singly entry and potentially multiple exits of the loops using standard compiler algorithms. Recognizing all induction variables in the binary, however, would require an interprocedural analysis that we have not implemented, so induction variables are not recognized. Note that machines employing stride value prediction will effectively recognize the induction variables and ignore most of their dependences. For machines with last value or no value prediction, the loop iteration counting code generated by a typical uniprocessor compiler will result in at least one data dependence across iterations that needs to be synchronized, even if the loop is otherwise parallel.

4. Speculative Loop Parallelism

Loop iterations are the traditional target of parallelization and an obvious candidate for thread-level speculation. Each iteration of a loop can be turned into a speculative thread that runs in parallel with the other iterations of that loop. The only form of control dependences shared between iterations are loop termination conditions, and the outcomes are highly skewed in favor of continuation. The remainder of the control flow in each iteration is independent; thus failure to predict a branch within an iteration does not affect other threads. The degree of parallelism available in a loop is governed by the presence of data dependences that cross loop iteration boundaries. If the iterations operate on disjoint sets of data, the degree of parallelism can be equal to the number of iterations of the loop. In the following, we first focus on the common model of applying speculation to one loop at a time. We then look at the performance and hardware implications of allowing multiple loops to speculatively execute in parallel.

4.1. Single-Level Loop Speculation

When we restrict speculation to a single loop in a nest, the critical decision is which loop in the nest to speculate on. There are two factors that need to be considered when selecting the best loop.

- *Degree of Parallelism*: there must be sufficient data independence between the iterations to achieve parallelism. If the iterations are totally independent (a DoAll loop), then the potential degree of parallelism is equal to that of the number of iterations. If there are dependences across iterations (a DoAcross loop), the degree of parallelism is dependent upon the ratio of the length of the recurrence cycle to the length of the iteration.
- *Parallelism Coverage*: If we parallelize an inner loop, then all the code outside will run sequentially. Thus, it may be desirable to choose an outer DoAcross loop with less parallelism over an inner DoAll loop if speculation can only be applied to one loop at a time. We refer to the percentage of code executed under speculative execution as the parallelism coverage. By Amdahl's Law, low parallelism coverage necessarily results in poor performance.

To select the best loop, we developed a separate trace-driven tool called MemDeps[15]. This tool presumes that only one loop in any given dynamic loop nest can be speculatively parallelized. (Note that loops in a dynamic nest need not be defined in the same procedure.) MemDeps evaluates the speedup for each of the possible choices, and chooses the best performing loop in each dynamic nest to compute the overall speedup. At the end of the MemDeps simulation, we calculate the overall frequency with which each loop was dynamically chosen as the best loop. These overall frequencies are then used to make static choices for loops in our simulations of the various STP machine models.

We evaluate the performance of the one-level loop speculation model on several variants of the optimal STP machine using the SPECint95 benchmark suite. Figure 2 presents the experimental results of this study. Machines are denoted by their synchronization policy and memory load prediction scheme, if any, as described in Table 2.

The largest speedup of 5.2 is achieved by `ijpeg`, an image compression program, with stride prediction enabled (Optimal-SVP). The significant performance improvement seen with stride prediction is due to the elimination of induction variable dependences across iterations. (Last-value prediction has no effect on these variables.) Had the code been compiled for an STP machine explicitly, the compiler would recognize many of these induction variables and would eliminate their dependences from the program. If an STP machine used induction variable

recognition but not general value prediction, its performance would be bounded by the Optimal and Optimal-SVP results.

It is not surprising that `ijpeg` performs well as its algorithm is very parallel. `M88ksim` and `vortex` are the only other programs with speedups over 2, with the rest of the benchmarks performing only between 1 and 1.6 times better than sequential execution. Note that `li` and `perl` are relatively unaffected by value prediction.

Despite the optimal STP machine's ability to speculate loops perfectly, the overall harmonic mean of speedup achieved across the benchmark suite is only 1.6. With the exception of `ijpeg`, the results are rather disappointing especially when considering that the optimal STP machine uses an unbounded number of processors, delays every operation optimally, and has zero-communication cost. Moreover, the loop choices are made by analyzing the execution of the program with the same input set. Unless large changes are made to the code, speculating at only one level in each loop nesting will not yield significant speedup on a realistic STP machine. Different code generation or instruction scheduling could provide a potentially higher limit, however.

To gain more insight into the performance results, we instrumented the code and the simulator to collect various characteristics of the individually speculatively parallelized loops[15]. We determine the computation time of the loops speculated on, whether the loops are innermost or outer loops, and whether the loops are DoAcross or DoAll loops. We summarize the findings of those experiments here.

- *Poor overall performance due to poor parallelism coverage.* Many of the programs have loops that show fairly impressive speedups when speculatively executed. However, the lack of parallel coverage, shown in Table 4, results in overall performance that is much lower. For example, `m88ksim` with stride prediction achieves an impressive 34.4-times speedup on 71% of the program, but the serialization in the other 29% of the computation drags down the overall performance. To illustrate the importance of coverage, we also present the Amdahl's Law limit on the overall speedups in Table 4, which assumes infinite speedup of the covered portions of the programs and sequential execution elsewhere. Even if the all the parallelized code was executed in a single cycle, most programs would still show relatively modest overall speedups.

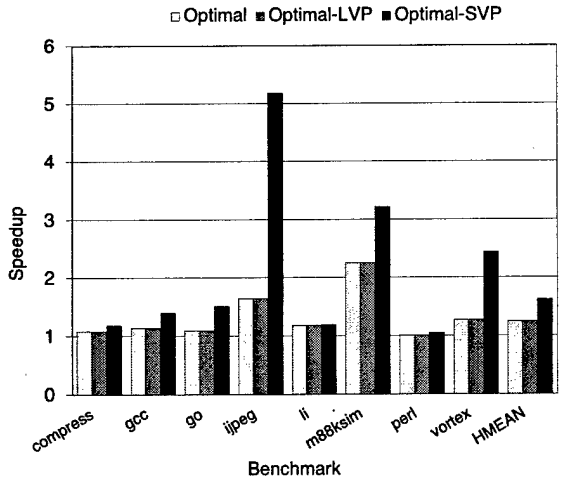

Figure 2. Optimal STP Speculation on Single-Level Loops

Table 4: Coverage and Maximum Theoretical Speedup for Single-Level Loops

Program	% Parallelized	Amdahl's Law Max Speedup
compress	27%	1.4
gcc	77%	4.4
go	67%	3.0
ijpeg	99%	100.0
li	45%	1.8
m88ksim	71%	3.5
perl	85%	6.7
vortex	93%	14.3

- *Trade-off between speedup and coverage.* In many cases, choosing the best loops for overall program speedup presented a distinct trade-off. Inner loops tended to have higher speedups but lower parallel coverage, while outer loops covered more of the program but had lower speedups.
- *Lack of DoAll loops.* Only the two best performing programs (`ijpeg` and `m88ksim`) spend more than 10% of their execution time in DoAll loops. Most of the integer programs have only DoAcross loops, which tend to have a lower degree of parallelism.
- *Size of the speculative state is limited.* In all but three of our benchmarks (the three being `li`, `vortex`, and `gcc`), the maximum amount of speculative state needed per thread was relatively small, approximately three kilobytes. Indeed, Steffan and Mowry found that threads needed less than 5KB of buffering on average (usually significantly less), and that a two-way set associative data cache with a small fully-associative victim cache was sufficient to retain almost all speculatively-accessed lines[19].

The analysis of the above suggests that it is important to find other sources of speculative parallelism. Doing so will not only increase parallelism coverage, but will also enable the machine to exploit parallel inner loops more effectively.

4.2. Multi-Level Loop Speculation

Speculatively executing multiple loops in a nest seems to be an obvious approach to improving single-level loop performance, but there are many difficulties in practice. First, the relatively small number of processors we are targeting (4 or 8 in our later experiments) make it difficult to assign them to multiple loops at a time. In addition, because the number of iterations in a loop is not always known *a priori*, some loops would occupy the entire machine with "potential" iterations. Finally, our thread prioritization favors the most-sequential threads, so inner loop threads would tend to force outer loop threads out of the machine.

Despite these difficulties, we wish to find a bound for this approach. In this experiment, we consider the extreme case where the optimal STP machine uses an unbounded number of processors to simultaneously execute all the iterations of each loop in a nest that it encounters. The results are shown in Figure 3.

Even with all its idealistic characteristics and its use of a very aggressive speculation model, the optimal STP machine's performance is still relatively poor. The harmonic mean improves only modestly from 1.6 to 2.6 (with stride prediction) when speculatively executing all loops simultaneously, while the machine design becomes much more difficult.

4.3. Summary of Speculative Loop Level Parallelism

Our results show that speculatively executing one loop at a time will not yield significant speedup under the STP model. Moreover the performance is highly sensitive to the way the code is written. Most integer codes have DoAcross loops which have limited parallelism. The ability to speculate on just one loop in each nest limits the parallel coverage which produces a lower overall speedup. Parallelizing multiple loops simultaneously increases the coverage and the overall performance, but would be very difficult to effectively support in a real machine. Overall, the performance on a very idealistic system is still modest. This result strongly suggests that loop-level speculation needs to be complemented with other sources of parallelism.

5. Procedure Level Speculation

Procedures are the programmer's abstraction tool for creating loosely-coupled units of computation. This suggests that it may be possible to overlap the execution of a procedure with the code normally executed upon the return of the procedure. A characteristic that makes speculative procedure execution particularly attractive is the lack of control dependence between the sequential and speculative threads. Procedures are expected to return under normal execution, and thus it is seldom necessary to discard the speculative work because of control flow. The only

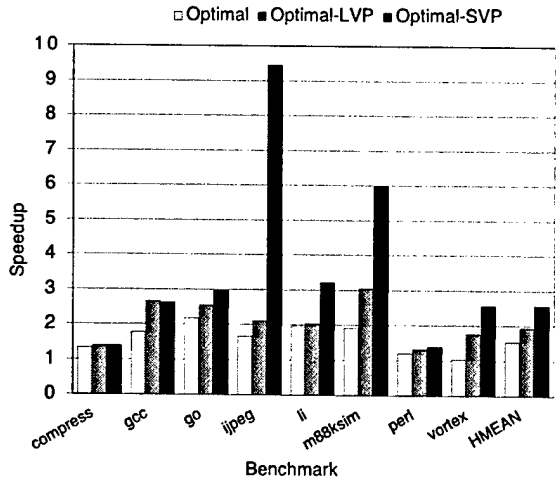

Figure 3. Optimal STP Speculation on Multi-Level Loops

exceptions are when the procedure raises an exception or uses unconventional control flow operations such as `longjmp`. These unusual circumstances can easily be handled by simply aborting the speculative threads and discarding their speculative states.

Unlike loops, procedures have not been a very popular target of parallelization. They have generally been used in more functional programming environments where there are fewer memory side effects in procedures and recursion is more common[5][7]. In typical imperative programming environments, procedures tend to share more data dependences with their callees. Also, as recursion is less predominant in imperative programs, the available parallelism is not scalable. These limitations, however, are much less important to the STP model. The speculative execution hardware can handle the memory dependences that might exist across procedures. Furthermore, a real STP machine is likely to have hardware support for only a small number of concurrent threads. The prevalence of procedure calls throughout programs provides a potentially effective source of parallelism for complementing loop-level parallelism.

To speculate at the procedure level in the STP model, we concurrently execute the called procedure with the code following the return of the procedure. Notice that it is the latter that executes speculatively. We propose to use a new thread to execute the called procedure and have the original thread execute the rest of the caller code speculatively. A data dependence violation occurs if the code following the return reads a location before the callee thread writes to that location. The same mechanism that is used for loop-level parallelism can be used to ensure that the data dependences are satisfied. By customizing the procedure calling convention to support speculation, the overhead of creating a new thread could be minimized. Furthermore, if the threads have their own private registers, there would be no need to save and restore registers at procedure boundaries. (Our study does not take advantage of this optimization opportunity.)

While loop level speculation can occupy an arbitrarily large number of processors by assigning a new iteration to each processor, each instance of procedural speculation creates work for only one additional thread. To create more opportunities for parallelism, this concept of procedural level speculation can be applied recursively. In the recursive case, note that the order of thread creation is not the same as the sequential order that the threads will retire in. The sequential ordering of the recursively created threads can be easily determined as follows. If thread A creates a speculative thread B at a call site, then B comes after A, and inherits from A all of its sequential ordering relationship with all other threads.

Because the return value is often used immediately upon the return of a procedure, speculatively executing the code following the procedure body could result in a large number of rollbacks. To avoid these rollbacks, we propose predicting the value that will be returned. Return value prediction is implemented by keeping a cache of past values returned by each procedure, if they exist. The caller thread continues to execute the code following the procedure call using the predicted return value. When the callee thread returns, the actual return value is compared to the predicted value. If the values are different, the machine would

generate a data violation signal, discard the speculative state, and restart the thread.

5.1. Predictability of Return Values

To verify that speculation with return value prediction has potential, we first look at the predictability of those values. We experimented with two simple schemes of prediction: last-value prediction and stride-value prediction. The results are shown in Figure 4. We classify procedure returns into three categories: (1) those that have either no return values or whose return values are not used, (2) those whose return values are used and are correctly predicted, (3) those whose return values are used and are mispredicted. For each program, we show two sets of data, one that uses last-value prediction labelled "L" and one that uses stride-value prediction labelled "S".

First, we observe that both last-value and stride-value prediction give similar results, with those of last-value prediction being slightly better for half of the programs. Misprediction of return values occurs less than 50% of the time for all programs, with `vortex` and `m88ksim` having almost no mispredictions. The benchmarks where return value prediction most often fails typically return pointers or other memory/storage related values. For example, `compress` makes many calls to a hash function whose results are highly unpredictable. Those that are extremely predictable tend to return quantities like status/error conditions, as in `vortex` for example. Finally, note that just because a return value is correctly predicted does not imply that much of the callee and caller computation will be overlapped; if the procedure modifies global variables or reference parameters after the caller has speculatively read such data, and the values read are not predictable, then the caller thread will be rolled back.

5.2. Evaluation of Procedural Speculation

Our next experiment evaluates the speedups of the procedural speculation model on optimal STP machines with different value prediction policies.

The results are shown in Figure 5. First, by comparing the performance of Optimal and Optimal-LRVP we observe that

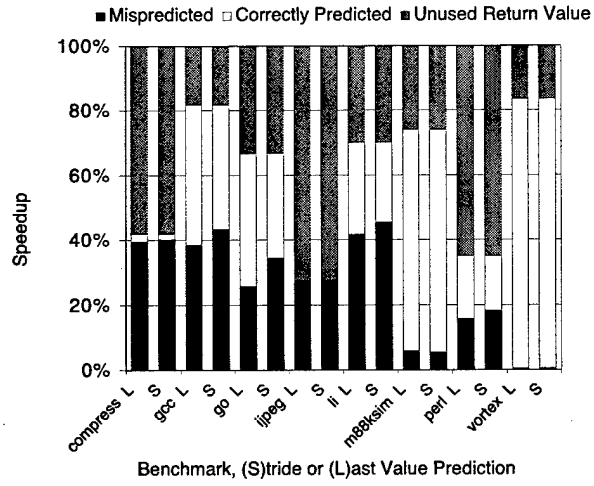

Figure 4. Predictability of Procedure Return Values

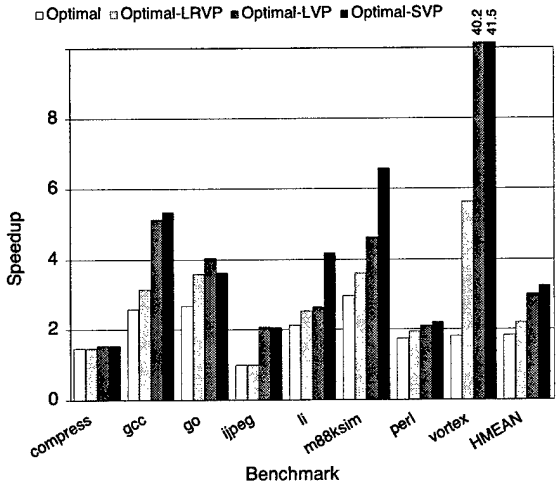

Figure 5. Optimal STP Speculation on Procedures

result value prediction has a significant positive effect on the performance of procedural speculation. As expected, programs with highest numbers of used and correctly predicted return values (`vortex`, `m88ksim`, as well as `go` and `gcc` to a lesser extent) benefit significantly. Conversely, `compress`, whose return values are not predictable, shows almost no improvement. `ijpeg` shows essentially no improvement with return value prediction because its most frequent routines (discrete cosine transforms) do not return any values.

Value prediction on regular data accesses is useful for almost all the programs, and can sometimes make a dramatic difference to the performance as in the case of `vortex` and to a lesser extent `m88ksim` and `gcc`. To gain some insight on this issue, we analyzed the code for `m88ksim`, a program that simulates a microprocessor. We found that the load instruction that benefits most from stride value prediction is a load of the simulation's program counter (at the beginning of the `datapath()` procedure). Since the program counter typically increments by 4 each time `datapath()` is called, stride value prediction is perfect for eliminating this dependence. Just as a processor benefits greatly from prefetching consecutive instructions, this speculative execution enables `m88ksim` to run much faster.

We also investigated the behavior of `vortex`, which has abundant parallelism when prediction is enabled. The dominant procedure in `vortex`, `Chunk_ChkGetChunk`, accounts for about 18% of the total execution time. The procedure verifies that a memory chunk access was valid and sets a call-by-reference parameter, `status`, to indicate the type of error if any. The return value is a boolean version of `status`. Given that the error conditions rarely occur, this is an excellent procedure for speculation. Note that prediction of both the return value and the call-by-reference out parameter is needed to make the threads completely parallel.

Overall, the experiments suggest that procedures are a good source of speculative parallelism in many programs. With the use of return value prediction, speculating at procedural boundaries delivers a performance comparable to that of executing all loops

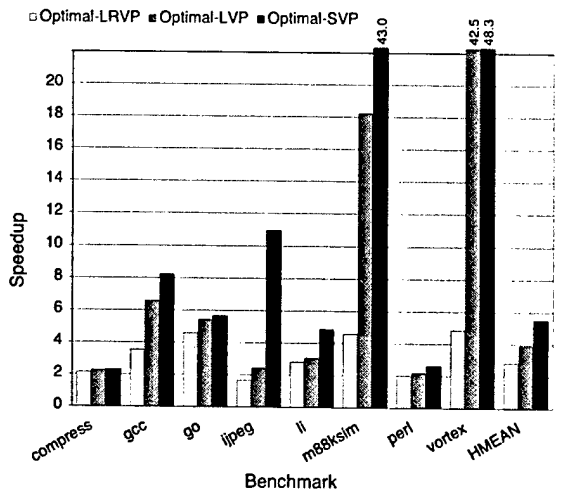

Figure 6. Optimal STP Speculation on Loops and Procedures

speculatively on the various Optimal STP models. Value prediction of regular memory accesses improves the overall speedup for almost all programs and has a major impact on specific programs.

6. Speculating at Both Procedural and Loop Boundaries

We next investigate the effect of combining both procedure and loop-level speculation. The experimental results for the various optimal STP models are shown in Figure 6, and they are much more encouraging. Most of the programs improve significantly over speculation on loops or procedures only, showing benefits from both forms of speculation. All but `compress` and `perl` have at least a 4.5-times speedup under the Optimal-SVP model. This includes programs, such as `gcc`, which have been very difficult to parallelize previously. As noted before, value prediction can have a significant effect on specific programs.

7. Experimenting with More Realistic Models

Having shown that speculation at all loop and procedure boundaries exposes a reasonable amount of parallelism in an optimal STP machine, we now experiment with this software model on more realistic machine models. In the following sections, we first evaluate the speculative scheme on an optimal STP machine with a finite number of processors, and then on the base STP machines that may require rollbacks.

7.1. A Finite Number of Processors

An optimal STP machine with an unbounded number of processors favors the creation of as many threads as possible. In the degenerate case where every single instruction is a thread of its own, the results would be identical to those reported by previous oracle studies where each operation is issued as soon as its operands become available[22]. Speculating at all procedure and loop boundaries can easily generate more threads than a reasonable implementation could maintain.

On a machine with support for only a finite number of threads, we must have a strategy to prioritize among the available threads. We adopt the simple strategy of prioritizing the threads according to their sequential execution order; a thread earlier in the sequential execution order has a higher likelihood of success and is thus given higher priority. In the presence of recursive procedural speculation, a newer thread may have a higher priority than an existing thread. When that happens, the machine frees up a processor for this thread by terminating the speculative execution of the thread with the lowest priority and discarding its speculative state. When the machine has some free resources, it will (re)start the execution of the thread with the highest priority. With this strategy, speculation on inner loops can occupy all available resources and thus prevent any speculative execution progress in the outer loops. In cases where the coverage or parallelism of the outer loop is more compelling, allowing inner loop speculation to "preempt" outer loop speculation would not be desirable. To address this, we suppress the speculation of all inner loops if they are estimated to have less parallelism than outer loops. We estimate the degree of parallelism of each individual loop by measuring the ratio of the average iteration length to the average length of the critical recurrence across iterations. This value is derived dynamically using the MemDeps simulator described in Section 4.1.

Figure 7 and Figure 8 show the performance achieved with 4-way and 8-way STP machines. As expected, the speedups are lower than those found with infinite processors. `M88ksim`, `vortex`, and `ijpeg` perform quite well, delivering over a 5-times speedup on an 8-way machine and roughly 3-times speedup on a 4-way machine (both with stride prediction). Value prediction continues to benefit the same programs that saw improvement in the infinite processor case, but the gains are much more realistic and limited. The program `compress` suffers little degradation, but its performance with infinite processors was quite low to begin with. Overall, the harmonic mean of the speedups is 3.2 for 8 processors, and 2.3 for 4 processors. Achieving a performance of 3 and 4 for the larger programs, `gcc` and `vortex`, respectively, on an 8-way machine is particularly encouraging.

7.2. Machines with Rollbacks

The most unrealistic aspect of the optimal STP machine is that it automatically delays every operation by the optimal amount, guaranteeing that there are no dependence violations to cause rollbacks. In this section, we present an experiment where we remove this fundamental assumption. We evaluate 4-way and 8-way machines that insert no delays into their executions, and upon detection of a dependence violation, must squash the thread and roll back the computation. This causes a performance degradation when the machine speculates on threads that try to read data before it is written and are unable to predict the value correctly.

In Figure 9 and Figure 10, we show the results for 4-way and 8-way Base STP machines, respectively. As expected, the performance of each Base machine is lower than that of the corresponding Optimal machine. Nonetheless, the results are surprisingly good given that the machines use no synchronization

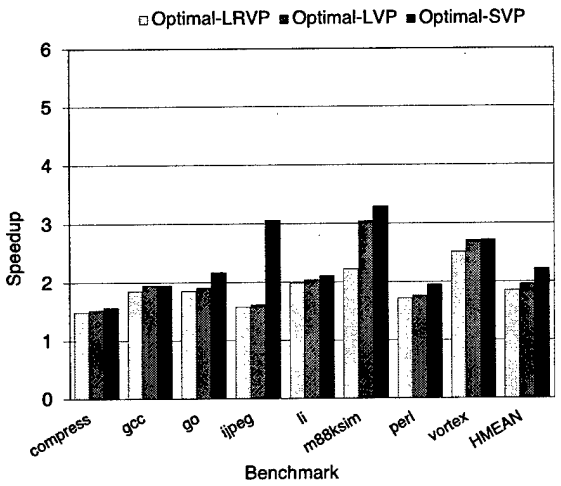

Figure 7. Speculating both Loops and Procedures on a 4-way Optimal STP Machine

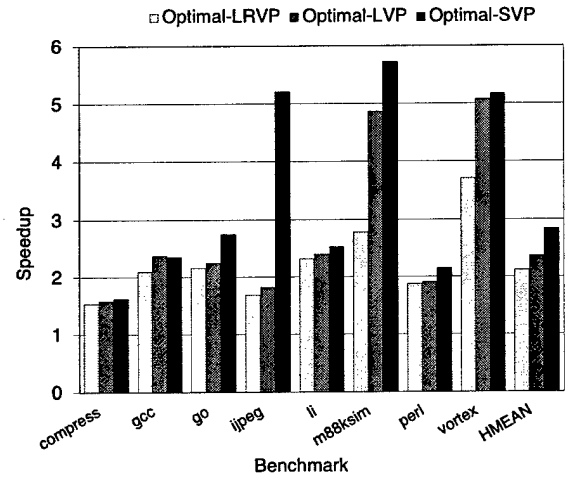

Figure 8. Speculating both Loops and Procedures on an 8-way Optimal STP Machine

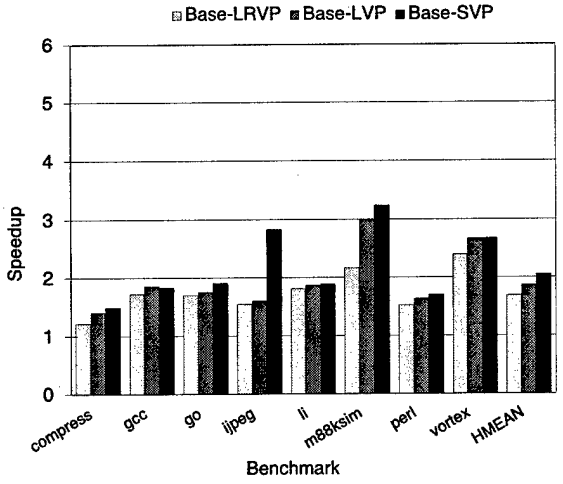

Figure 9. Speculating both Loops and Procedures on a 4-way STP Machine with Rollback

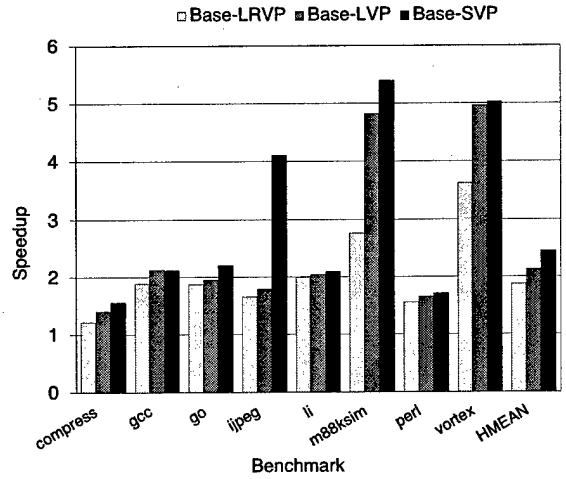

Figure 10. Speculating both Loops and Procedures on an 8-way STP Machine with Rollback

operations at all to minimizing rollbacks. Harmonic mean speedups range from 1.7 to 2.1 for a 4-way base machine depending on the value prediction employed.

It should be noted that the finite-processor Base STP machines are still rather idealistic. For example, the communication between threads incurs no overhead and thread creation and rollback is instantaneous. While more realistic models would result in lower performance, there are still many optimizations that could improve the performance as well. Implementing additional techniques to suppress poorly performing threads could provide a significant benefit. Other possibilities include introducing static or dynamic synchronizations[13] into the program in order to reduce the number of rollbacks. Further research on specific software and hardware mechanisms is necessary to effectively harness this form of parallelism.

8. Related Work

The STP model is based on the Multiscalar paradigm which was the first speculative thread architecture[3][17]. The Multiscalar is a complete architecture for executing speculative threads with register and memory forwarding mechanisms and a mechanism for detecting memory data dependence violations called the address resolution buffer (ARB). The Multiscalar research group has evaluated the base processor and extensions to the processor that avoid unnecessary thread restarts using a number of integer applications, showing moderate speedups[13]. The Multiscalar group and other researchers have augmented the cache-coherency mechanisms of a single chip multiprocessor to support speculative threads [4][13][17]. The goal of these approaches is to achieve lower hardware overheads and more flexibility than the ARB approach originally proposed in the Multiscalar processor. To select tasks for the Multiscalar[21], a compiler pass examines

the control-flow graph of the program and uses heuristics to partition the basic blocks into threads. Speculative tasks are supposed to immediately follow the spawning task, so there is no nested task creation, but prediction of successor tasks is more difficult.

Steffan and Mowry evaluate the performance potential of a multiprocessor-based approach and show reasonably good performance on some integer applications[19]. Unfortunately this performance seems to be quite dependent upon extremely aggressive and idealistic dynamic instruction scheduling.

The Trace processor is a concrete machine proposal that can exploit similar parallelism found in multiple flows of control[16]. In the Trace processor the program is dynamically broken into a sequence of instructions, each of which can be executed speculatively by a separate thread of control. If an instruction violates a data dependence, only that instruction and the instructions dependent on it will be re-executed. The ability to selectively re-execute only those instructions that are affected mimics the ability of an oracle that can execute every instruction optimally whenever its operands are ready. Unfortunately, this ability comes with a relatively high implementation cost, as the processor must keep track of enough information to recover from all combinations of mispredictions. This necessarily constrains the size of each thread—the proposed maximum thread length is 16 instructions. This limitation prevents the system from exploiting parallelism with a larger granularity. In comparison, the STP machine model can realistically allow longer speculative threads than that of the Trace processor because there is only one speculative state per thread. It can exploit parallelism between instructions that are farther apart, and can follow more independent flows of control because threads are explicit. However, compared to the Trace processor, STP thread restarts are expensive so it is more important to minimize dependence violations.

The Dynamic Multithreading (DMT) processor[1] combines features of the Simultaneous Multithreading machine[20] and the Trace processor while also supporting procedural and some loop speculation. It executes threads in a tightly-coupled SMT-like environment. Selective recovery is performed from a trace buffer like the trace processor, but the machine does not compact normally executed code into traces. The speculative thread size is limited by the need to keep all of the thread's dynamic instructions in the trace buffer; in their simulations, threads can be at most 500 instructions long. They chose loop and procedure continuations as their targets for speculative execution. Inner loops speculation is not supported in order to work with preexisting binaries. While our earlier work showed that speculating only on single-level (mostly inner) loops in integer codes is insufficient, inner loops are still a valuable contributor to performance in certain programs. The DMT machine also does not perform explicit return value prediction. The only prediction employed is for registers, and it is limited to predicting that the initial register values for a child thread are the same as the parent thread's register values at the time the child is spawned. The DMT processor does have a fast-forwarding mechanism for when register prediction fails, though there is no facility for memory load prediction or synchronization. We believe that even greater performance is possible if these speculative support features are strengthened.

Hammond, Willey, and Olukotun present a sample machine design implementing many of the ideas suggested in this paper and elsewhere, but the results reported are not encouraging[6]. There are many factors which contribute to this, however. First, to simplify the implementation, thread management is done largely in software by handlers that take significant time to operate. For example, starting a loop, committing a loop iteration, and ending a loop all take on the order of 70 to 80 instructions, which is fairly close to the average thread size that we observed. Additionally, while return value prediction is implemented for procedures, neither value prediction nor synchronization is implemented to help cope with the other data dependences across speculative threads. Also, communication between threads occurs in the L2 cache and costs 10 cycles. A subsequent paper by the group showed improved results when overheads were reduced and software was manually updated to introduce synchronization and better code scheduling[14]. The reduced overheads were achieved by abandoning procedural speculation, however. Our study is intended to find the potential speculative thread-level parallelism in programs. Whether this parallelism can be efficiently exploited by a real machine is outside the scope of this paper. The results from Hammond et. al. would suggest that efficient speculation support is necessary in order to achieve good performance.

General motivation for using multiple flows of control to increase sequential application performance was presented by Lam and Wilson[9]. While the value prediction we employ could result in performance beyond the dataflow limit observed in Lam and Wilson's experiments, our multiple flows of control (loop iterations and procedures) are much more restricted in nature.

9. Summary and Conclusions

We summarize our search for speculative parallelism with Figure 11, which shows the harmonic means of the performance results of the different experiments we performed. We started our exploration by assuming an optimal STP machine with an infinite number of processors that completely avoids rollbacks. We experimented with different speculation schemes: speculating only one loop at a time, speculating at all loop levels, speculating at all procedural boundaries, and finally to speculating at both loop and procedure boundaries. We found that the last scheme delivers impressive performance on the optimal STP machine. Having found such a scheme, we then considered more realistic machine models. We first refined the parallelization scheme to reduce the number of threads created, and evaluated the performance of the programs on optimal STP machines with 8 and 4 processors. Finally, we experimented with base machines that roll back speculative threads whenever dependence violations are detected.

The methodology used in this paper enabled us to analyze programs effectively and discover promising sources of speculative parallelism. The relaxed machine model (Optimal) allowed us to quickly identify the fundamental limitations of loop

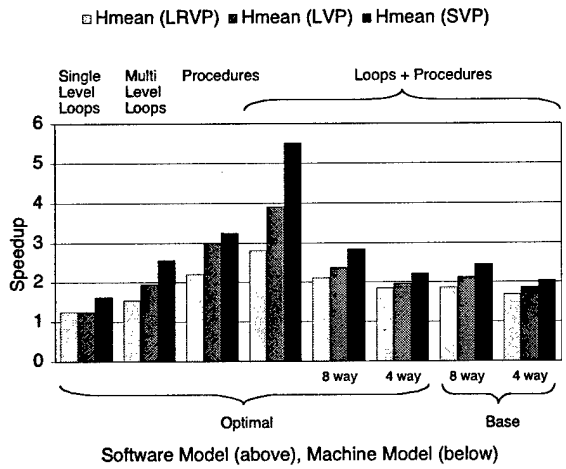

Figure 11. Summary of the Harmonic Mean Speedups

level speculation. We were able to develop a variety of analysis and simulation tools that isolated parallelism coverage as an important factor in the lack of performance. This result led us to locate alternative sources of speculative parallelism, namely procedural speculation. The gradual refinement of the machine models from the optimal STP machine, first with an infinite number of processors, then to a finite number of processors, and finally to machines with rollbacks increased our understanding of the different factors that affect performance.

This paper shows that the combination of loop and procedural speculation (with result value prediction) is a promising parallelization scheme for speculative thread-level parallel machines. This scheme achieves at least a 4.5-times speedup for six of the eight SPECint95 programs on the optimal STP machine with an infinite number of processors, a 2.4 times speedup on an 8-way machine that rolls back, and a 2.0 times speedup on a 4-way machine. While much research remains to be done to define suitable hardware mechanisms, to develop new software optimization techniques and to calculate evaluate the effectiveness of specific systems, our results suggest that STP is a potentially effective technique for speeding up general-purpose applications.

References

[1] H. Akkary and M. A. Driscoll, "A Dynamic Multithreading Processor," *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pp. 226-236, Dallas, TX, November-December 1998.

[2] R. Cytron, "Doacross: Beyond Vectorization for Multiprocessors," *Proceedings of the International Conference on Parallel Processing*, pp. 836–844, 1986.

[3] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 58-67, Gold Coast, Australia, May 1992.

[4] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi, "Speculative Versioning Cache," *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, pp. Las Vegas, NV, 1998.

[5] R. H. Halstead, "Multilisp: a Language for Concurrent Symbolic Computation", *ACM Transactions on Programming Languages and Systems*, 7(4):501-538, October 1985.

[6] L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pp. 58-69, San Jose, CA, October 1998.

[7] T. Knight, "An Architecture for Mostly Functional Languages," *Proceedings of the ACM Lisp and Functional Programming Conference*, pp. 500-519, August 1996.

[8] V. Krishnan and J. Torrellas, "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor," *Proceedings of the 1998 ACM International Conference on Supercomputing*, pp. 85-92, Melbourne, Australia, July 1998.

[9] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 46-57, Gold Coast, Australia, May, 1992.

[10] J. R. Larus, "Estimating the Potential Parallelism in Programs," *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, pp. 331-349, MIT Press, 1991.

[11] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 138–147, Cambridge, MA, October 1996.

[12] P. Marcuello, A. González, and J. Tubella, "Speculative Multithreaded Processors," *Proceedings of the 1998 ACM International Conference on Supercomputing*, pp. 77-84, Melbourne, Australia, July 1998.

[13] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 181–193, Denver, CO, June 1997.

[14] K. Olukotun, L. Hammond, and M. Willey, "Improving the Performance of Speculatively Parallel Applications on the Hydra CMP," *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[15] J. Oplinger, D. Heine, S. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun, "Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors," Computer Systems Laboratory Technical Report CSL-TR-97-715, Stanford University, February 1997.

[16] J. E. Smith and S. Vajapeyam, "Trace Processsors: Moving to Fourth-generation Microarchitectures," *Computer*, vol. 30, pp. 68–74, September 1997.

[17] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425, Ligure, Italy, June 1995.

[18] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 196–205, Orlando, FL, June 1994.

[19] J. G. Steffan and T. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, NV, February 1998.

[20] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 392-403, Ligure, Italy, June 1995.

[21] T. N. Vijaykumar and G. S. Sohi, "Task Selection for a Multiscalar Processor," *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pp. 81-92, Dallas, TX, November-December 1998.

[22] D. W. Wall, "Limits of Instruction-Level Parallelism," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 176-188, Santa Clara, CA, 1991.